Streaming SIMD Extensions -- 3D Transformation

Version 1.3

1/99

Order Number: 243631-004

Table of Contents

Revision History

Revision	Revision History	Date
1.3	FCS update.	01/99

References

The following documents are referenced in this application note, and provide background or supporting information for understanding the topics presented in this document.

1 *Increasing the Accuracy of the Results from the Reciprocal and Reciprocal Square Root Instructions using the Newton-Raphson Method,* Intel Application Note AP-803, Order No: 243637-001.

1 Introduction

Streaming SIMD Extensions include Single Instruction Multi-Data (SIMD) instructions. The basic data type used in Streaming SIMD Extensions is the packed-single precision floating point data type. This paper shows how to write efficient code with Streaming SIMD Extensions to perform transformation operations. The focus of this paper is to show how the transformation and lighting operations work together efficiently, and how to use new prefetch instructions to reduce memory load latencies.

This paper discusses two kinds of data structures: the traditional Array of Structures (AoS), with data organized according to vertices - $x_0\ y_0\ z_0$, and the Structure of Arrays (SoA), with data organized according to coordinates - $x_0\ x_1\ x_2\ x_3$. The SoA data structure is the more natural structure for SIMD instructions. The original C code is based on the Intel 3DR code.

The `\samples\3d_trans\` subdirectory contains C and assembly files with Streaming SIMD Extensions and a small test program in C, which you can build and run.

2 Transformation and Lighting

The transformation with projection basically converts the vertices from the object space to the screen space with respect to the specific viewer properties.

The following figure shows the basic transformation structure:

$$\begin{bmatrix} Transform \\ Matrix \end{bmatrix} * \begin{bmatrix} Object \\ Space \\ Vector \end{bmatrix} = \begin{bmatrix} Screen \\ Space \\ Vector \end{bmatrix}$$

The traditional vertices representation uses the AoS format. When using the AoS format not all the data elements of Streaming SIMD Extensions are used. A better format for the Streaming SIMD Extensions is the SoA format. Both methods are faster than scalar floating-point code.

2.1 Implementing Transformation

This section provides some guidelines on the implementation of transformation and lighting projections in 3D transformation applications. In addition, two new features introduced by the Streaming SIMD Extensions are presented, enabling additional performance gain.

2.1.1 Improving the Transformation and Lighting Pipeline

In most cases the vertices data exceed the L1 cache and reduce performance. The data is loaded from main memory each time it is transformed or lit. For example, in the case of five passes on each vertex, the data is loaded five times from main memory.

To eliminate unnecessary memory traffic, perform all the operations while the vertices data is in the L1 cache. This can be done by performing all the lighting and transforming in one pass. Then, perform all the operations for one vertex (transformation and all light sources), and only then move on to the next vertex. However, this approach is efficient only when you use one fixed lighting configuration.

In order to enable several lighting configurations, you need to perform several function calls for each vertex. A preferable approach is to perform the transformation and lighting in blocks. First, divide your vertices data into blocks of 128 vertices each (in the case of a 16 K byte data-cache). Next, perform the transformation and lighting on the same block. After completing all processing on this block, start processing the next block.

2.1.2 Two SIMD Approaches to Transformation and Lighting

This paper discusses two ways of arranging the vertices data. The traditional method is the AoS arrangement, with a structure for each vertex. However this method does not take full advantage of the Streaming SIMD Extensions. The best processing method for code using Streaming SIMD Extensions is to arrange the data in an array for each coordinate. This data arrangement is called SoA. This arrangement allows more efficient use of the parallelism of Streaming SIMD Extensions, because the data is ready for transformation. Another advantage of this arrangement is reduced memory traffic, because only the relevant data is loaded into the cache. Data that is not relevant for the transformation (such as: texture coordinates, color, and specular) is not loaded into the cache.

There are two options for transforming data in AoS format. One is to perform SIMD operations on the original AoS format. However, this option requires additional calculations. In addition, some of the operations do not take advantage of the four SIMD elements in the Streaming SIMD Extensions. Therefore, this option is less efficient. The recommended way for transforming data in AoS format is to temporarily transpose each set of four vertices to SoA format before processing it with Streaming SIMD Extensions.

The following is a simplified transposing example:

Original format:

```
x1,y1,z1,   x2,y2,z2,   x3,y3,z3,   x4,y4,z4,
```

Transposed format:

```
x1,x2,x3,x4   y1,y2,y3,y4   z1,z2,z3,z4
```

Most Streaming SIMD Extensions require 16-byte alignment of the data. Therefore, you may have to force all the relevant data elements to 16-byte alignment.

These are the data structures for both methods:

AoS data structure:

```
typedef struct{
    float x,y,z;
    int color;
    . . .
}Vertex;
Vertex Vertices[NumOfVertices];
```

SoA data structure:

```
typedef struct{
    float x[NumOfVertices];
    float y[NumOfVertices];
    float z[NumOfVertices];
    int color[NumOfVertices];
    . . .
}VerticesList;

VerticesList Vertices;
```

2.1.3 Interface to the Lighting Routine

While performing transformation and lighting, the transformation routine can pass the data to the lighting routine in an intermediate SoA format. For example, some applications require the external input/output to the transformation and lighting routines to be in AoS format. In this case, you need to transpose the AoS input into SoA format once, then perform the transformation and all the light sources' calculations in SoA format. Finally, de-transpose back to the AoS format to create the output.

Figure 1 shows the vertices format at the different stages of the routine.

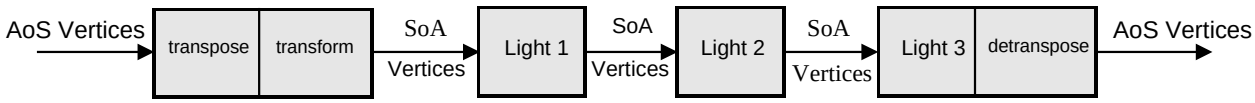

Figure 1: Changing AoS Format to SoA Format

2.1.4 Tips and Tricks

This section presents two additional Streaming SIMD Extensions features for improving the performance of your program.

- Data prefetching instructions
- Divide approximation

2.1.4.1 Data Prefetching

The Streaming SIMD Extensions introduce new prefetching instructions. Use these prefetching instructions to reduce the latency of loads from main memory. These instructions do not affect the normal functionality

of the program. However, in some boundary cases, prefetch does affect program behavior. The prefetch instruction brings the entire cache line of the specified data closer to the processor, in cache hierarchy. The prefetch instruction does not block program execution and minimizes interference with program execution. You should be aware that prefetch still requires processor resources for execution and does retire, though the retirement mechanism has been optimized.

The `aos` example (provided in the `\samples\3d_trans\` subdirectory) uses the prefetch instruction to bring the input vertices into the L1 cache. At the beginning of each iteration, the next iteration's data is prefetched. The output vertices memory area is not prefetched, due to the non-temporal nature of the output vertices. However, note that the use of streaming stores in this case may minimize cache pollution. In this code the new prefetch instruction improved the performance of the kernel by an estimated 30%.

2.1.4.2 Divide Approximation

The `divps` instruction performs an accurate division operation, however it has a ~ (35) cycle latency and throughput for the Pentium® III processor.

The Streaming SIMD Extensions introduce the `rcpps` instruction that approximates a 1/x operation. The `rcpps` instruction has a latency of 2 and throughput of 1, which is a significant improvement over the `divps` instruction. The `rcpps` instruction error is $1.5 * 2^{-12}$. In some cases this error margin can be acceptable, such as for a lighting algorithm. However, in some cases, better precision is necessary and the Newton-Raphson Method is quite useful, such as for this 3D transformation algorithm. In some other cases, we need to use the `divps` instruction. The choice is dependent on the algorithm.

An efficient way to receive a more accurate result, without paying the long latency of the `divps` instruction, is to use the Newton Raphson algorithm (NR):

```
Y0 = rcpps(X)
Y1 = 2 * Y0 - X * Y02
```

The following code implements one iteration of the NR algorithm.

```
rcpps    xmm1, xmm3
movaps   xmm5, xmm1
addps    xmm5, xmm5
mulps    xmm1, xmm1
mulps    xmm1, xmm3
subps    xmm5, xmm1
```

One iteration of NR doubles the accuracy of the `rcpps` result. The cost is five low latency instructions. In cases where this approximation is not acceptable, use the `divps` instruction.

3 Performance

The performance of a 3D-transformation application can be improved through the use of several of the Streaming SIMD Extensions features as described in the next sections.

3.1 Gains/Improvements

The Streaming SIMD Extensions enable increased performance over scalar floating-point code, through utilizing the SIMD feature of these instructions. When the data is arranged in SoA format, one instruction handles four data elements. This arrangement also eliminates loading data that is not relevant for the transformation, such as texture coordinates, color, and spectral information.

Additional performance gain is achieved by prefetching the data from main memory, and by replacing the long latency `divps` instruction with a low latency `rcpps` instruction, or its Newton-Raphson approximation for better precision. For more information, refer to the Intel Application Note AP-803, *Increasing the Accuracy of the Results from the Reciprocal and Reciprocal Square Root Instructions using the Newton-Raphson Method*, Order No: 243637.

3.2 Considerations

3.2.1 AoS and SoA

The best performance is achieved by performing the transformation with data in SoA format. However some applications require the data in AoS format. In these cases it is still possible to use Streaming SIMD Extensions, by transposing the data to SoA format before the transformation and lighting operations. After these operations are complete, de-transpose the data back to AoS format.

3.2.2 Avoiding Dependency Chains when using movlps/movhps

In some processor implementations, the `movhps/movlps` instruction bypasses one part of the register. In the worst case, this creates a dependency chain that links successive iterations of the loop, even if those iterations are otherwise independent. Parallelism is lost, resulting in inferior performance. Therefore you should avoid writing code that contains chains of dependent calculations.

The dependency problem can occur with the `movhps/movlps/shufps` sequence, since each `movhps/movlps` instruction bypasses part of the destination register. These instructions cannot execute until prior instructions that generate the corresponding register are completed. This dependency can prevent successive loop iterations from executing in parallel.

One solution to this problem is to include a 128-bit load from a dummy local variable (such as "brk_dep" in the attached code) to each register used with a `movhps/movlps` instruction. This effectively breaks dependency by performing an independent load from a memory or cached location. In some cases, such as loading a section of a transform matrix, the code that uses the swizzled results already includes 128-bit loads. In these cases, an additional explicit 128-bit dummy load is not required.

4 Conclusion

The Streaming SIMD Extensions provide several features that enable increased performance of 3D transformation applications. These are the main reasons for performance gain in applications using Streaming SIMD Extensions:

- SIMD feature of these instructions.
- `prefetch` instruction enables reducing memory latency.
- `rcpps` instruction with an NR algorithm can replace the long latency `divps` instruction, in cases where the full precision of the `divps` instruction is not necessary.

To reduce memory traffic, divide your vertices list into blocks that fit into your L1 cache. Perform all the operations on one block (transform, first light, second light etc.) before starting operations on the next block.

When using the Streaming SIMD Extensions, there is an advantage in storing the data in SoA format. These are the main reasons for this speedup:

- Fewer operations. When using the SoA code, the data is ready for the calculations. There is no need to transpose the data before the calculations.
- Reduced memory traffic. The data in the AoS format is interleaved with data that is not used in the transformation, such as texture coordinates, color, specular data. This unused data is copied to the cache. When using the SoA format, only the necessary data is copied to the cache.

To achieve maximum acceleration arrange your data in SoA format. If, for any reason, you cannot use the SoA format, use the AoS format, and transpose the data before transforming. After completing all the transformation and lighting operations, de-transpose the data back to AoS format.

5 SoA Code Examples

Arranging data in SoA format is the most efficient approach for performing transformation and lighting with Streaming SIMD Extensions. The `\samples\3d_trans\soa\` subdirectory contains the C and assembly code files with data in SoA format discussed in this application note.

`\main.c`	C code implementation.
`\soa.asm`	Assembly language implementation with Streaming SIMD Extensions.
`\build.bat`	Example of a `.bat` file that builds an executable file.

5.1 C Code for SoA Transformation

The following C code performs transformation on data arranged in SoA format.

```
/* Perspective projection and transformation */
void TransformProjectSoA(VerticesList *inp, VerticesList *out, int count,
camera *cam)
{
int i;
float x,y,z;
float orw;

for (i=0; i<count; i++){

    x = inp->x[i], y = inp->y[i], z = inp->z[i];

    orw = x * mat->_30 + y * mat->_31 + z * mat->_32 + mat->_33;
    out->x[i] = (x*mat->_00 + y*mat->_01 + z*mat->_02 + mat->_03)*(cam->sx/orw)
          + cam->tx;
    out->y[i] = (x*mat->_10 + y*mat->_11 + z*mat->_12 + mat->_13)*(cam->sy/orw)
          + cam->ty;
```

```
      out->z[i] = (x*mat->_20 + y*mat->_21 + z*mat->_22 + mat->_23)*(cam->sz/orw)
            + cam->tz;
      out->w[i] = orw;
     }
  }
```

5.2 Assembly Code for SoA Transformation

This section shows an example of transformation of data in SoA format, using Streaming SIMD Extensions. This method is the most efficient way to perform transformation and lighting with Streaming SIMD Extensions. This routine passes through the vertices and processes four vertices at a time. The functionality of this code is equivalent to the C code provided in the code example. These are the routine parameters:

`inp`	pointer to the input vertices area
`outb`	pointer to the output vertices area
`count`	number of vertices
`cam`	the camera properties

The `_matSoA` is the transformation matrix global variable, in which each element is replicated four times. The size of the arrays of the SoA is assumed to be constant. Before starting the loop (lines 69-73) the program loads the pointers from the stack to Pentium® III xmm registers, and sets the loop counter (`ecx`) such that [`eax+ecx`] points to the last four X input values, and [`edx+ecx`] points to the last four X output values. At the beginning of the loop (lines 77-79), the program loads four elements of X, four elements of Y and four elements of Z into three xmm registers. In lines 82-91 the program calculates four output W elements. The next step is to calculate 1/W for multiplication of the viewport scale later in the loop. In lines 95-107, the program calculates four X output elements. In lines 110-122 the program calculates four Y values, and in lines 125-137 the program calculates four Z values.

This example uses the fast-approximated reciprocal instruction instead of the accurate but slow divide. For games, this approximation (ERROR <= $1.5 * 2^{-12}$) is acceptable. See Section 2.1.4.2 for information on more accurate divide approximations.

5.3 Optimized SoA Transformation

The sample assembly code is an optimized example of transformation of data in SoA format, using Streaming SIMD Extensions.

In the optimized code the instructions are rescheduled to expose more parallelism to the processor. The basic code is composed of four independent blocks, inhibiting parallel execution. The instructions in each block are data dependent. In the following optimized code the instructions of each two adjacent blocks are interleaved, enabling much more parallel execution.

This optimization is made under the assumption that the vertices data is already in the cache. If the data is not in the cache, this code becomes memory bound. In this case, try to add more computations within the loop, for example, lighting calculations. Another option is to prefetch the data, using the Streaming SIMD Extensions prefetch instruction.

6 AoS Code Examples

The traditional data arrangement is AoS format. To take advantage of the Streaming SIMD Extensions when transforming and lighting data in this format, the program must change the format to SoA temporarily. The SoA arrangement allows the most efficient transforming and lighting with Streaming SIMD Extensions.

The `\samples\3d_trans\aos\` subdirectory contains the C and assembly code files with data in AoS format discussed in this application note.

`\main.c`	C code implementation.
`\aos.asm`	Assembly language implementation with Streaming SIMD Extensions.
`\build.bat`	Example of a `.bat` file that builds an executable file.

6.1 C Code for AoS Transformation

The following C code performs transformation of data in AoS format. The functionality of this code is equivalent to the assembly code in Section 0.

```
/* Perspective projection and transformation */
void TransformProject(InVer *inp, OutVer *out, int count, camera *cam)
{
int i;
float x,y,z;

float orw;

for (i=0; i<count; i++){
   x = inp[i].p.x, y = inp[i].p.y, z = inp[i].p.z;

   orw = x * mat->_30 + y * mat->_31 + z * mat->_32 + mat->_33;
   out[i].p.x = (x*mat->_00+y*mat->_01+z*mat->_02+mat->_03)*(cam->sx/orw)+cam-
        >tx;
   out[i].p.y = (x*mat->_10+y*mat->_11+z*mat->_12+mat->_13)*(cam->sy/orw)+cam-
        >ty;
   out[i].p.z = (x*mat->_20+y*mat->_21+z*mat->_22+mat->_23)*(cam->sz/orw)+cam-
        >tz;
   out[i].p.w = orw;
}
```

6.2 Assembly Code for AoS Transformation

The AoS transformation assembly code is based on the SoA transformation assembly code. In this code, each set of four AoS vertices are transposed to the SoA format (lines 85-102), then the SoA vertices are transformed in the same way they as in the SoA routine. In this code, after the transformation the output vertices are de-transposed back to the AoS format. When you perform lighting after the transformation, use the vertices from the transformation routine in SoA format, without transposing and de-transposing between passes.

Lines 87, 94, 102, 114 provide examples of a prefetch instruction.